The code you are seeing is the result of a long, ugly development process. I started the project about a month ago, thinking that a head start would be perfect – I would give myself all the time I needed to think through a good puzzle-making algorithm, I’d have plenty leftover for the DLB backend, and then I could tweak anything I needed to. I started with gusto, and then I just let it hang over my head until this weekend.

To say that fixing the half of a program that I had was difficult would be an understatement. I remembered how I wanted to implement the solution, generally speaking, but I was too early in the development to avoid writing new code, but too late in development not to debug things. In short, it was a pain and a half, and when I finally scraped together something that seemed to be working, it was slow, ugly, and I wasn’t proud of anything about it. It was with this that I wrote the DLB backend, which in comparison was a very nice, simple implementation that only took about an hour. Running larger tests revealed that somehow my code (despite having taken nearly a month to write), was missing cases. Like, hundreds of thousands of cases. I was distressed.

So I spent most of Sunday and Monday dreading when I eventually would turn in the program, knowing that it wouldn’t work. Around 6pm, I decide that I’d rather put together something new and see if just doing it all in one go would be better, so I sat down, and wrote. After two grueling hours of remembering what I did before and more or less rewriting it (but using better coding practices this time with regards to abstraction and commenting), it actually worked! Not only that, but it solved the problem exactly the same way as before, but correctly! Unfortunately, as of the time of this writing, there are three and a half hours to the submission deadline, so any execution times you see greater than that are going to be more or less guesses. But hey, it works!

So down to the important bit – implementation. The puzzle itself is represented by a 2-Dimensional char array (upper left hand corner is [0][0] and lower right hand being [N-1][N-1]). This ensures that we use exactly as much room as we need for the puzzle and no more - as opposed to using, say, an arraylist or a class, which would cause more overhead. The algorithm starts at coordinates (0, 0), and very simply moves right, dropping to the next row, and setting x to 0 when it gets too far. In order to check the validity of any particular square, it grabs all the letters to the west and north of the square (using another nice bit of recursion), and checks the two against the dictionary. I do check one before the other though, because if left doesn’t work, there’s no use in checking up. I chose to examine the left word first because we’ll be able to check longer words on average with left, and it’s a little faster to check longer words with the DLB implementation (because fewer long words just happen to be prefixes for other words).

So the program checks all the letters for the square, and if the square is valid, it goes to the next square and starts checking there. Any time that a square makes it through all the letters, it returns, going back to the previous square. Because of the magic of recursion, that square picks up right where it left off: if it was trying ‘p’ on the square, when we get back to it, it moves on to ‘q’. This is pruning at its very best – no wasting time with previous letters. In this way, we run from the beginning of the board to the end and back again, like a big game of “Red Rover.” This finds all solutions by covering every valid permutation of the board.

There are obvious differences between the run times with DLB vs MyDictionary. Because MyDictionary uses a linear search, the worst case is that it has to search the entire dictionary to find the word you’re looking for, and has to evaluate every letter of every word until you find it (though that would require some really, really tricky finangling on the part of the dictionary author). The DLB has the nice effect that the maximum number of steps is the depth of the tree, which means the length of the longest word, though on average our word will be shorter than that. Runtimes vary greatly when one algorithm depends on the size of the dictionary and the other doesn’t.

Admittedly, it does take setup time to create the DLB, whereas we can just hit the ground running with the MyDictionary. There are only two cases I can think of where MyDictionary beats out DLB. For one, if our dictionary was a small, stratified list of words, and we were dealing with lots of long keys. Second, MyDictionary takes up no extra room, whereas DLB must create a node for each unique letter sequence in the Trie. However, in most modern cases (which involve long, similar dictionaries and more space than an astronaut set adrift), DLB is the clear choice.

|  |  |  |
| --- | --- | --- |
| Runtime Analysis Table (time to find 1 element) | | |
| Test File | MyDictionary (seconds) | DLB (seconds) |
| 3a | < 1 | < 1 |
| 3b | 2.15 | < 1 |
| 4a | 2 | < 1 |
| 4b | 5 | < 1 |
| 4c | < 1 | < 1 |
| 4d | < 1 | < 1 |
| 4e | 300 | 17 |
| 4f | 8 | < 1 |
| 5a | 5 | < 1 |
| 6a | 600 | 360 |
| 6b | 1500 | 5 |
| 6c | 1800 | 68.19 |
| 7a | 86400 | 7200 |
| 8a | 172800 | 7200 |
| 8b | ---------- + a lot | ----------- |
| 8c | 900 | 3.71 |

\*Note: Estimated values may be wildly inaccurate

Tested Values

Untested, Estimated Values